Tools for Monitoring and Controlling Distributed Applications

N91-CR 15 P18

Keith Marzullo*
Mark D. Wood**

TR 91-1187
February 1991

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593, Contract N00140-87-C-8904. The views, opinions and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy or decision. This work was also supported by a grant from Xerox.
**This author was also partially supported by a G.T.E. Graduate Student Fellowship.

Tools for Monitoring and Controlling Distributed Applications

Keith Marzullo*
marzullo@cs.cornell.edu

Mark D. Wood†
wood@cs.cornell.edu

Cornell University
Department of Computer Science
Ithaca, New York 14853
November 16, 1990

Abstract

The Meta system is a UNIX-based toolkit that assists in the construction of *reliable reactive systems*, such as distributed monitoring and debugging systems, tool integration systems and reliable distributed applications. Meta provides mechanisms for instrumenting a distributed application and the environment in which it executes, and Meta supplies a service that can be used to monitor and control such an instrumented application. The Meta toolkit is built on top of the Isis toolkit; they can be used together in order to build fault-tolerant and adaptive distributed applications.

*This work was supported by the Defense Advanced Research Projects Agency (DoD) under NASA Ames grant number NAG 2-593, Contract N00140-87-C-8904. The views, opinions, and findings contained in this report are those of the authors and should not be construed as an official Department of Defense position, policy, or decision. This work was also partially supported by a grant from Xerox.

†This author was also partially supported by a G.T.E. Graduate Student Fellowship.

1 Constructing Reactive Systems

In a *reactive system* architecture, the system is partitioned into two pieces: an environment that follows a basic course of action, and a control program that monitors the state of the environment in order to influence the environment's progress. This architecture is very general. For example, process control systems, system monitors and debuggers, and tool integration services all have a reactive system structure.

Another application of the reactive system architecture is the structuring of distributed applications. For example, many distributed applications are constructed by taking off-the-shelf programs and connecting them with some communication subsystem. Such an application can be thought of as an "environment" with a state including the properties of machines running the application, current performance of the component programs, and the state of the communication subsystem. The job of the control program is to monitor the state of the application in order to guarantee that the system operates efficiently in spite of changing load and failures. The control program can also be used to interconnect the application's components in a more loosely bound manner than conventional RPC mechanisms.

The Meta system, described in this paper, is a UNIX[1]-based toolkit that provides the basic primitives needed to build a non-real-time reactive system. Using the toolkit, a distributed program can be instrumented with sensors and actuators in order to expose its state for purposes of control. Meta provides mechanisms that allow a control program to query the state of the instrumented application and to respond by invoking actuators when some condition of interest occurs. The toolkit includes facilities for structuring individual components into collections of components for fault-tolerance. In addition, Meta guarantees that the monitoring and reaction is done atomically.

Meta itself is built on top of another toolkit, the Isis system. The application designer can use Isis for fault-tolerant communication and Meta for distributed control. In fact, the Meta project was started when four of us in the Isis project worked on integrating a distributed application constructed from off-the-shelf components [MCWB90]. The facility we found lacking in Isis was support for distributed control.

The next section introduces the architecture of an application managed by Meta. Section 3 presents how applications are instrumented, and Sec-

[1] UNIX is a trademark of A.T.&T.

tion 4 discusses how the resulting application is controlled. Finally, Section 5 presents the current status of Meta and discusses our future plans.

2 The Meta Architecture

The architecture of Meta can be illustrated through an example of managing a distributed application. Consider an application that includes services and clients making use of the services. A given service consists of a set of identical servers replicated both for fault-tolerance and for coarse-grained parallelism. Meta will be used to manage the services; in particular, if the load on a service is too large or the number of servers becomes too small due to crashes, then a new server is to be started and added to the service. Additionally, if a server's queue becomes too long, then waiting requests are to be migrated to less-loaded servers in the service. There are other conditions that would probably need to be maintained as well, such as reducing the number of servers when appropriate, but for sake of brevity we will keep our example limited.

Meta structures a distributed application using a data model based on the entity-relation data model [Che76], with each instrumented component (*i.e.*, a program equipped with sensors and actuators) being viewed as an *entity* and its sensors and actuators being the *attributes* of that entity. For example, a server in the above example could be instrumented with sensors that give the server's load and the queue of waiting requests. Entities of the same type, that is, having the same set of sensor and actuator attributes, form an entity set.

Subsets of an entity set may be grouped together to form *aggregates*. Aggregate structures provide control programs with a way of grouping related entities together and limiting actions to members of that group. For example, the servers comprising a service can be grouped into an aggregate representing the service. Aggregates are themselves entities, and the system architect can define sensors and actuators on aggregates. An aggregate sensor is a function over the state of all the members of the aggregate. For example, a service aggregate could have a sensor that gives the median queue length of the servers in the service. An aggregate actuator causes an action to be performed on some subset (from one to all) of the current members.

A distributed application is managed through the use of guarded commands; that is, through a set of (*condition, action*) pairs that reference the sensors and actuators of the instrumented application. These commands

are executed by interpreters that reside in *stubs* (somewhat like RPC stubs) coresident with the instrumented programs, thus allowing for fast notification and reaction. Each condition is a proposition on the state of system; references to both local sensors—within the entity to which the stub is attached—and nonlocal sensors are allowed. The action portion is a sequence of actuator invocations that are executed atomically. Actions may enable guarded commands on another Meta stub; this facility allows one to write control programs that span multiple components.

Since guarded commands are evaluated in the same address space as an instrumented program, their impact on the performance of the application is a concern. The syntax of the guarded command language (a postfix language called NPL) is tailored for fast and efficient evaluation, and so we do not expect programs to be written directly in this language. We are designing an object-oriented control language called *Lomita* [MCWB90] that can be used to describe the structure of the application and to specify its control behavior. A Lomita program contains a schema specifying the entity and aggregate structure along with their sensors and actuators. The control behavior of the application is specified in Lomita through the use of rules, where the conditions for the rule may include real-time interval logic expressions [SMSV83]. Such temporal expressions are compiled into finite state automata, where the state transitions are implemented using Meta guarded commands.

Figure 1 illustrates the use of stubs. The machine M_1 is running a server that has been instrumented, so there is a stub running in the same address space as this server that can directly access the sensors and actuators of the server. The machine is also running a separate Meta-supplied program accessing the various properties of the machine and its operating system, such as the amount of available memory and the processor load. This program is instrumented, and so has a stub that supports a set of sensors and actuators over the machine and operating system state.

3 Application Instrumentation

An application first must be instrumented before it can be controlled. This is accomplished by inserting into the application a small amount of code, and then linking the application with a Meta library. This section describes the instrumentation process in more detail.

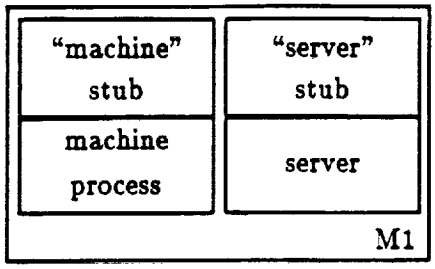

Figure 1: An Instrumented Component

3.1 Access to Base Values

A sensor provides access to the value of some underlying system variable. An application defines a sensor with a Meta library routine:

```
meta_new_sensor(svr_q_length, "load", TYPE_INTEGER, min_period);
```

This routine creates an integer-valued sensor named "load". When this sensor is referenced, the function svr_q_length in the instrumented program is called, which presumably returns the number of entries on the server's work queue.

In a reactive system, the fact that a sensor's value has changed is as important to know as the current value of the sensor. There are two methods by which an application can alert its stub that a sensor's value has changed. In some cases, a sensor's value changes either slowly or regularly, in which case a lower bound on the time between changes in its value can be determined. The application tells the stub this lower bound as the fourth parameter of the meta_new_sensor call. This value states how long that sensor's value can be cached before repolling is needed. In other cases, it would be very hard to determine such a lower bound. In this case, the fourth parameter of the meta_new_sensor call is zero, and the stub will obtain a fresh value only when the application makes an upcall to the stub. Such upcalls never block and can be made even when a nonzero polling period has been specified.

Actuators provide the means through which Meta acts upon the system. Like sensors, actuators are implemented by function calls in the application program. Actuators can be parameterized and can return either *success* or *failure*.

3.2 Functional Composition

A control program may wish to monitor a sensor whose value is a function of an existing sensor or sensors. For example, the control program may wish to monitor the maximum load of a server or the difference between two queue lengths supported by a server. Such sensors can be easily defined using Meta. A stub can construct functions of the sensors it supports and can define additional sensors in terms of these functions. The stub ensures that the sensors comprising such a sensor are sampled atomically. A extensive collection of pre-defined functions are available, and this collection can be augmented with user-defined functions.

3.3 Aggregates

An aggregate has, as predefined sensors, set-valued versions of the sensors on the components comprising the aggregate. For example, if a component has an integer sensor named load, then an aggregate of this component has a group sensor named load whose type is "set of integers" and whose value is the set of loads of the components. Other aggregate sensors can then be defined as functions of group sensors.

Just as an aggregate inherits the sensors of its components, an aggregate also inherits the actuators of its components. For example, if a component has an actuator named run, then an aggregate of this component has a group actuator named run. An invocation of the group actuator run invokes all of the component run actuators.

3.4 Fault-Tolerance

When necessary, sensor fault-tolerance is achieved through replication. The process containing the sensor to be made fault-tolerant is replicated, and the replicas are grouped into an aggregate; the value of the fault-tolerant, aggregate sensor is then a function of the members' sensor values [Sch90].

The severity of sensor failures that can be tolerated depends on the choice of aggregate function. For example, to provide tolerance to crash failures, the aggregate function need only pick one of the member's values to return as the sensor value. In this case, the availability of the sensor is the same as the availability of any member of the aggregate. In process control systems, however, a real-world sensor such as the temperature of a reaction vessel can be represented as an interval bounding the actual value of the quantity

being measured. In this case, a fault-tolerant intersection function can be used to mask arbitrary failures of sensors [MW90,Mar90].

Group actuators are useful for achieving fault-tolerance in that they can be used to implement *coordinator-cohort* based actuation [ISI90]. When invoking a group actuator, the command can include two additional parameters: an integer specifying the number of individual actuations to perform, and a preference list of aggregate members which indicates which aggregate members to try first. If the chosen actuator fails, then another member will be picked according to the preference list until the number of requested actuations is achieved or can not be achieved, in which case the group actuation fails.

4 Control

Once an application is instrumented, a control program can be written. The basis for controlling applications in Meta is a language of guarded commands that reference the state of the instrumented application.

4.1 Interpreting Guarded Commands

Each Meta stub implements a guarded command interpreter that has direct access to the sensors and actuators of the component to which the stub is attached. A stub can reference sensors and actuators not local to the component by communicating with the interpreter that does have direct access. The name of a sensor or actuator is sufficient for the Meta system to resolve which interpreter has direct access. So, a guarded command can be executed by any stub, although some stubs would provide better performance than others.

Since aggregates are not represented by a single component in the application, some stub must be selected to maintain the definitions of a given aggregate's sensors (and actuators). Exactly which stub computes the aggregate values is up to the application designer; either an existing stub or a "Meta server" (a stub instrumenting a dummy process) can be designated to do so, and other stubs can be designated as *cohorts*[2] that will take over in case the stub instrumenting the aggregate fails. This approach central-

[2]These cohorts should not be confused with the cohorts in the ISIS *coordinator-cohort* facility, although the concept is the same. We are currently investigating how to best implement this structure.

izes the computation of aggregate values, which in turn facilitates providing consistent views of the aggregate's state.

The interpreters for Meta guarded commands may also be made fault-tolerant through replication. In this case, one interpreter is responsible for executing a given guarded command while the others remain as standbys. Sufficient state is exchanged among the replicas so that one of the standbys can take over in case the primary interpreter fails.

In our client-server example, the servers of a service are grouped into an aggregate. Each member of the aggregate (a server) has been instrumented, as described previously in Section 3, with a sensor that gives the load of the server. An aggregate sensor can then be defined that provides some measure of the service load, such as the median load of all the servers. If each server is equipped with an actuator that accepts a request for migration, then reliable migration can be implemented by invoking the set-valued aggregate actuator with the number of actuations specified as one and the preference list selected, for example, from the servers' loads. The stub that implements the aggregate sensors and actuators could be one of the servers in the service (presumably in the server stub) or a separate Meta server.

4.2 Atomic Guarded Commands

Recall that a guarded command consists of a set of (*condition*, *action*) pairs. A condition is a propositional expression over the sensor values, and an action is a sequence of parameterized actuator invocations. Ideally, Meta would ensure that the action is executed as an *atomic command*, that is, atomically and consistently with respect to its triggering condition [LS84].

When a predicate becomes true, the action should be executed in the same state in which it was triggered, but due to the asynchrony in the environment this can not be done without introducing blocking. Instead, Meta guarantees that any reference to sensor values during the action sequence obtains the same value as when the condition was triggered. Another property of atomic actions is that either all of the action is executed or none of it is executed. Providing this property requires a transactional facility with the ability either to undo the effects of partial actions or to invoke a forward recovery mechanism. Additionally, to provide consistent execution, the intermediate states of the action should not be visible to other guarded commands.

Meta currently provides only a limited amount of atomicity. For example, if a guarded command references only the sensors and actuators of a

single component (either simple or aggregate), then its execution will be atomic. This amount of consistency is all that is needed for our client-server problem. For example, Meta will guarantee that if a machine is selected and removed from a *free-machine* aggregate when starting a new server, then the selection and removal will be done atomically (in this case, by using the coordinator-cohort facility of ISIS). Other applications will require stronger guarantees of atomicity, however, so we are currently examining mechanisms that will enforce stronger guarantees of atomicity when necessary.

4.3 Example

Figure 2 shows part of a Lomita description of our client-server application. The description first defines the schema for server entities. In this simplified presentation, a server contains separate actuators for starting and stopping a job, with jobs being named by a string. For the sake of discussion, we assume that a job may be started and stopped repeatedly. The service aggregate has the sensor sload which is defined to be the median load of the individual sensors. The run actuator starts a job on some member of the aggregate, and the preference list specifies that the member should be selected on the basis of its load.

The two rules shown in this figure are compiled into NPL programs. The first rule states that a job should be migrated from a server whose load is too high. This rule can be translated into a single guarded command that can run in the server's stub. The following C call distributes the NPL command to all server entities:

```
meta_npl("server",
        "load 5 > GUARD jobs First 'job' BIND job suspend
        job service('JobService').run");
```

This guarded command contains the conditional predicate *load* > 5 and then the action sequence of binding the variable job to the first job on the job list, suspending that job, and then resubmitting it for execution by invoking the service aggregate operator run.

The second rule is more complex; it states that if the size of a service is too small or the load remains high for too long, then a new server should be started. The Lomita compiler would translate this rule into a finite state automaton, which in turn would be implemented by a set of Meta guarded commands.

```
server: entityset
   attributes
      key name : string;
      sensor load: integer;
      sensor jobs: {string};
      actuator stop(string);
      actuator start(string);
   end
end

service: server aggregate
   attributes
      key port : string = "JobService";
      sensor sload : integer = median(load);
      actuator run(job : string) = start(job)[load,1,"<="];
      actuator create = ...;
      ...
   end
end

when server(Name).load > 5 do
   job = First(server(Name).jobs);
   server(Name).suspend(job);
   service("JobService").run(job);
end

when SIZE(service("JobService")) < 3 or
   during service("JobService").sload > 5 for 60
      always service("JobService").sload) > 5
do
   create(...);
end
```

Figure 2: Job Service

5 Discussion

The Meta project has explored the feasibility of toolkit-based architecture for building reactive systems and has applied this approach to distributed application management. Meta provides a uniform way of interconnecting disparate components, facilitating both the design of new systems and the construction of systems glued together from existing applications. Our approach has the benefit of separating management policies from their implementation—that is, how those policies are carried out.

5.1 Related Work

Although much work has been done on system monitoring, our work differs in that it combines control with monitoring to provide the general architectural support needed to construct a class of reactive systems. A prominent example of a system designed strictly for monitoring is the work of Snodgrass [Sno88]; in his work, the system state is cast as a temporal database. Systems for debugging (especially those for debugging distributed systems), are a specialization of general monitoring systems. These systems provide a way to access the system state and to watch for certain predicates to be satisfied through the use of breakpoints [MH89,Bat88]. Of particular interest is the system IDD [HHK85] that permits interval logic expressions in specifying breakpoints.

Lomita is a rule-based language built on a real-time extension of interval logic. The rule-based language we have found most similar to Lomita is L.0 [CCNS90]. However, this executable language does not deal with the problem of instrumenting existing applications nor does it use a sensor-actuator data model. Configuration systems such as Conic [KMS89] overlap with the use of Meta for distributed application management in that they facilitate interconnecting components, but they lack the means for specifying reactive behavior.

5.2 The ISIS System

Much of Meta depends upon facilities provided by the Isis toolkit. One such facility is the notion of a *group*. An Isis group is a named dynamic set of processes. Each member of the group has the same view of which processes are currently in the group despite other processes asynchronously joining the group, leaving the group and crashing. Among other uses, Meta uses

ISIS process groups to implement atomicity of aggregate invocation and to organize the members of an aggregate.

Providing consistent behavior in Meta relies heavily upon the notion of *virtual synchrony* provided by the ISIS system [BJ87]. The ISIS system make asynchronous events such as message receipts and group membership changes appear to happen synchronously. This property greatly facilitates reasoning about system behavior and constructing a system that behaves in a consistent manner. Fundamental to this property is the notion of an ordered broadcast. ISIS provides two important broadcast primitives [JB89]: *abcast*, which totally orders the broadcasts to a group, and *cbcast* which partially orders the broadcasts to a group dependent on the causal order of the broadcasts. For example, if two apparently concurrent events occur in the instrumented application, Meta can impose a global total order on these events by using *abcast*.

5.3 Status

Several iterations of prototypes have been built with the latest being available from Cornell as part of the ISIS toolkit. Work is currently underway on a major release supporting the complete functionality described here. Preliminary performance figures from this work show the system to impose a low amount of overhead. The following benchmarks were obtained by running Meta on Sun 4/60's with interprocess communication handled by ISIS over a 10 Mbps Ethernet.

The time to execute a simple guarded command of the form A GUARD B with trivial local sensor A and trivial local actuator B is 84.1 microseconds, with uncertainty less than .1 microsecond. This implies approximately 12,000 guarded commands can be executed a second.

The bulk of the time for remote actions is of course in the message delivery. The ISIS causal broadcast (cbcast) takes 14.4 milliseconds[3]; the ISIS atomic broadcast abcast takes up to twice as long. Running the previous simple guarded command at a remote interpreter takes 32.6 milliseconds. This figure includes one cbcast to the interpreter to report the value and an abcast from the interpreter to effect the actuation.

The act of referencing a remote sensor has some initial start-up cost, which we call the *subscription* cost. Upon receiving a subscription request

[3]Performance figures of the order of milliseconds are accurate to within .2 milliseconds with a confidence of 95 percent, except for the time to subscribe, which is accurate to within 1.1 milliseconds.

from some remote interpreter, a Meta stub will report all changes in the sensor's value to the subscriber. To get a feel for the subscription cost, we measured the time need to do the following: send to the local interpreter a guard that immediately triggers and causes the interpreter to subscribe to a remote sensor, get the first value, and cancel the subscription. This time was measured to be 66.2 milliseconds. This figure includes the time to parse the guard, but the cost of this should be negligible, less than one percent. Note that the guard is sent locally via cbcast rather than via a (faster) direct procedure call because we wish to support replication of interpreters. The cbcast therefore results in communication with the ISIS protocol server for that machine.

Note that all communication in Meta goes through the ISIS protocol server, a separate process running on each machine. Newer versions of ISIS now under development allow for restricted types of broadcasts to be sent directly to the intended recipients, bypassing the ISIS protocol servers. This results in considerable savings; a cbcast of this form only costs 5.6 milliseconds. The bypass mode of communication requires the sender and receiver to be in the same group, which is not typically the case in Meta. However, the current implementation of Meta does put aggregates in the same group, opening the way to use the bypass mode of communication, and we are currently exploring ways of exploiting it even further.

Previous versions of Meta have been released, but these did not support the complete NPL language but instead had the notion of a watch, in which a Meta stub could be instructed to wait for the value of some sensor to satisfy some relation. This earlier work has emphasized the benefit of detecting conditions as close as possible to the site at which they become satisfied.

We are currently building a network manager as a test application for Meta, and are designing a debugging and monitoring tool and a system configuration system.

5.4 Directions

The current Meta toolkit is adequate for use in systems in which timing is not crucial. Although guarded commands can make temporal assertions, given the potentially unbounded latencies in the underlying UNIX and Isis platforms, such assertions can only be viewed as approximate upper bounds. However, the structure that Meta provides is general enough that we should be able to extend it to real-time reactive systems as well.

There are two main obstacles we see to extending Meta to real-time sys-

tems. The first has to do with the underlying Isis toolkit; to guarantee bounded reaction time, the underlying causal broadcast and group membership protocols must provide some real-time guarantees. A companion project in the ISIS group is currently looking into structuring Isis under Mach to provide these two protocols. The second obstacle has to do with the semantics of guarded commands. Guarded commands currently have the semantics of atomic actions; if a guarded command is continuously enabled, then it will eventually execute. We need to add an upper bound on how long the command can be enabled without executing, and then build a scheduler that either guarantees the command will be executed within its deadline or aborts the command if it cannot be executed within its deadline.

Acknowledgements Several people have contributed to the Meta project. Nancy Thoman designed and wrote the first version of the guarded command language, and Wanda Chiu designed a reactive relational database that provided a testbed for earlier versions of Meta. Kenneth Birman and Robert Cooper have contributed much to the design of the overall system. We would also like to thank Robert Cooper and Laura Sabel for their helpful comments on earlier drafts of this paper.

References

[Bat88] Peter Bates. Debugging heterogeneous distributed systems using event-based models of behavior. In *SIGPLAN/SIGOPS Workshop on Parallel and Distributed Debugging*. ACM, 1988.

[BJ87] Ken Birman and Thomas Joseph. Exploiting virtual synchrony in distributed systems. In *Proceedings of the Eleventh Symposium on Operating System Principles*, pages 123–138. ACM SIGOPS, 1987.

[CCNS90] E. J. Cameron, D. M. Cohen, L. A. Ness, and H. N. Srinidhi. L.0: A language for modeling and prototyping communications software. Technical Report ARH–015547, Bellcore, April 1990.

[Che76] P. P.-S. Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[HHK85] Paul K. Harter, Dennis M. Heimbigner, and Roger King. IDD: An interactive distributed debugger. In *Proceedings of the Fifth International Conference on Distributed Computing Systems*, pages 498–506, 1985.

[ISI90] Cornell University, Department of Computer Science, Upson Hall, Ithaca, New York 14853. *ISIS - A Distributed Programming Environment - User's Guide and Reference Manual*, March 1990.

[JB89] Thomas Joseph and Kenneth Birman. *Reliable Broadcast Protocols*, pages 294–318. ACM Press, New York, 1989.

[KMS89] Jeff Kramer, Jeff Magee, and Morris Sloman. Constructing distributed systems in Conic. *IEEE Transactions on Software Engineering*, SE-15(6):663–675, June 1989.

[LS84] Leslie Lamport and Fred B. Schneider. The "Hoare logic" of CSP, and all that. *ACM Transactions on Programming Languages and Systems*, 6(2):281–296, April 1984.

[Mar90] Keith Marzullo. Tolerating failures of continuous–valued sensors. Technical Report TR 90-1156, Cornell University, September 1990.

[MCWB90] Keith Marzullo, Robert Cooper, Mark Wood, and Ken Birman. Tools for distributed application management. Technical Report TR 90-1136, Cornell University, June 1990. Submitted for publication.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4), December 1989.

[MW90] Keith Marzullo and Mark Wood. Making real-time reactive systems reliable. In *Proceedings of the Fourth ACM SIGOPS European Workshop*, pages 1–6. ACM SIGPLAN/SIGOPS, September 1990.

[Sch90] Fred B. Schneider. The state machine approach: A tutorial. *Computing Surveys*, 22(3), September 1990.

[SMSV83] R. L. Schwartz, P. M. Melliar-Smith, and F. H. Vogt. An interval logic for higher-level temporal reasoning. In *Proceedings of the Second Symposium on Principles of Distributed Computing*, pages 173–186. ACM SIGPLAN/SIGOPS, 1983.

[Sno88] Richard Snodgrass. A relational approach to monitoring complex systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.